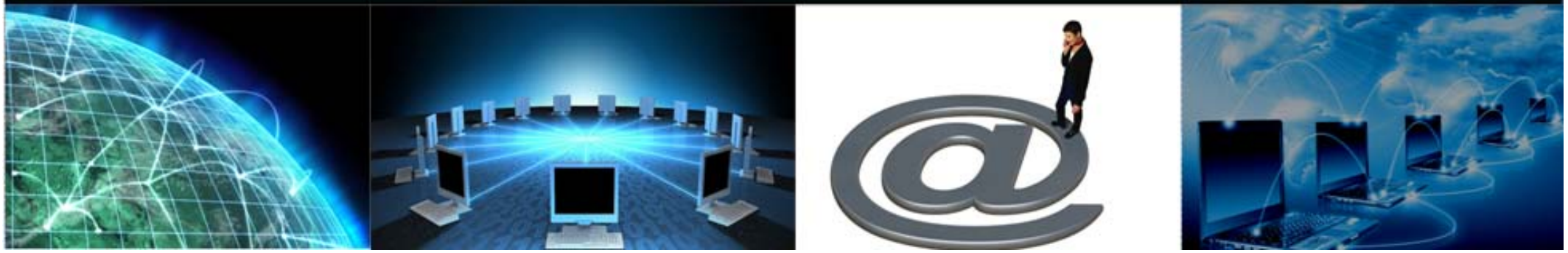


软件质量保证与测试

其他白盒测试方法

程序插装

- **程序插装（Program Instrumentation）是一种基本的测试手段，在软件测试中有着广泛的应用。**
  - 在调试程序时经常采用在程序中设置断点或打印输出语句，在执行过程中了解程序的一些动态特性。
- **这种方式。这就相当于在运行程序以后，一方面检验测试结果数据，另一方面借助插入语句给出的信息了解程序的动态执行情况。**

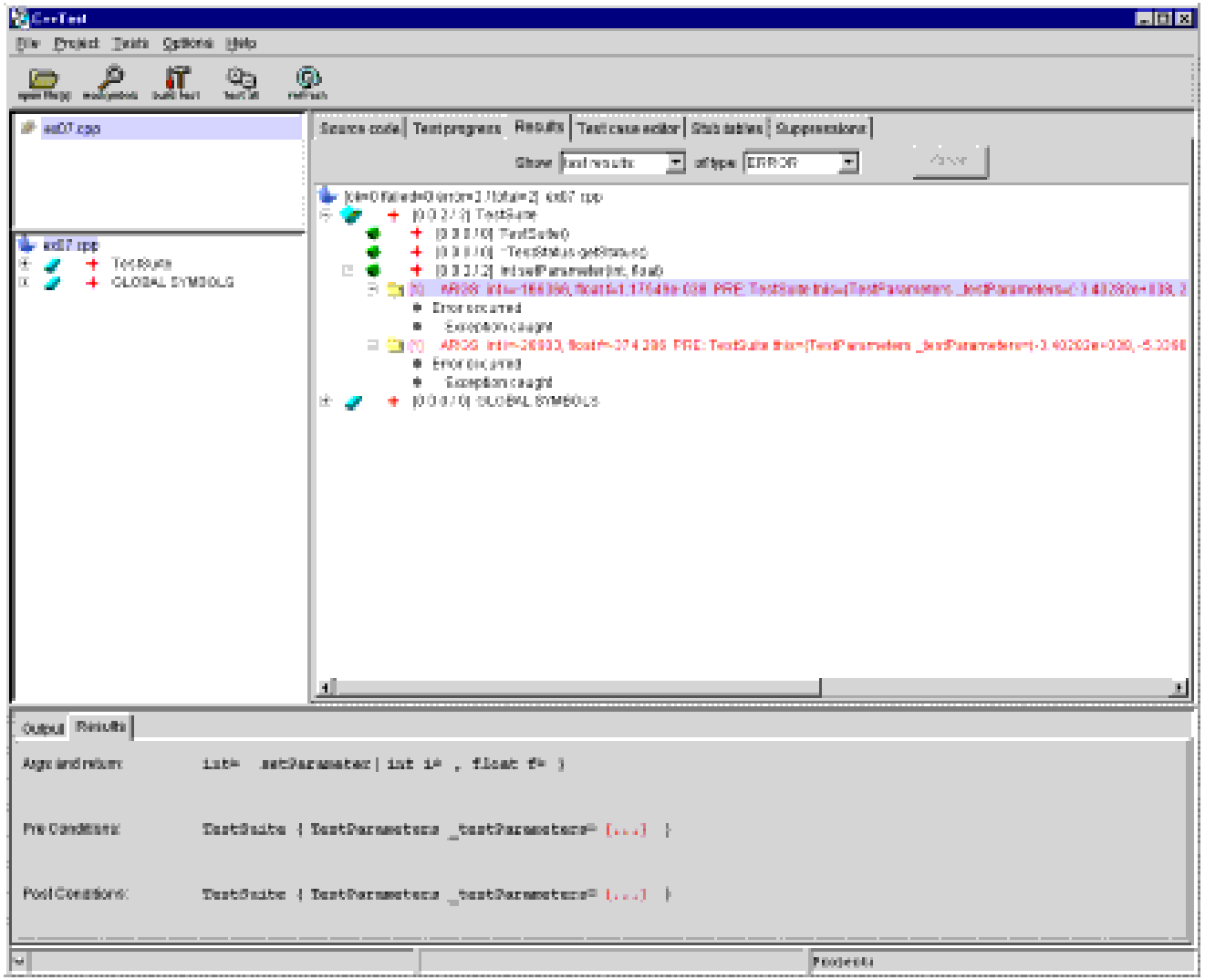
程序变异测试

- **假设P在测试集T上是正确的，可以找到P的变异体的某一集合：M＝{M（P）|M（P）是p的变异体}，若变异体M中每一元素在T上都存在错误，则可以认为源程序P的正确程度比较高，否则若M中某些元素在T上不存在错误，则可能存在以下三种情况：**
  - 这些变异体与源程序P在功能上是等价的。
  - 现有的测试数据不足以找出源程序P与气变异体的差别。
  - 源程序P可能含有错误，而某些变异体却可能是正确的。

白盒测试工具

- **C++Test介绍**
  - C++Test介绍
  - C++Test的单元测试功能
  - C++Test主要特性
    - 在不需要执行程序的情况下识别运行时缺陷
    - 自动化代码分析以增强兼容性
    - 优点

执行自动白盒测试



白盒测试工具JUnit

- 简介
- 特点
  - 使用JUnit的好处
  - JUnit测试编写原则
  - JUnit的特征
- JUnit使用

DoubleAdd.java

DoubleAdd.java

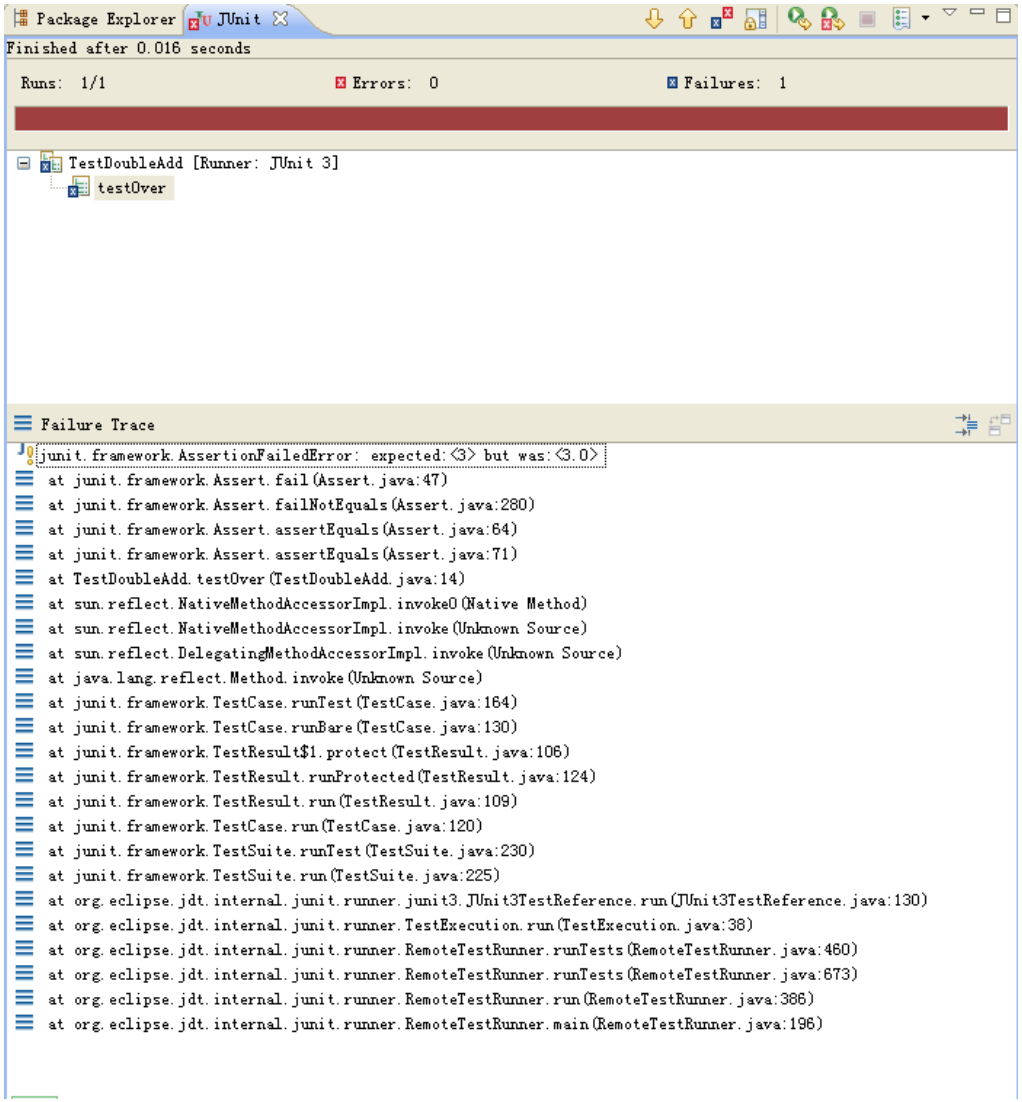
```
public class DoubleAdd {

    /**
     * @param args
     */

    public double add ( double number1 , double number2 ) {
        return    number1 + number2 ;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        DoubleAdd number = new DoubleAdd();
        System.out.println(number.add(1, 2));

    }

}
```

测试用例

DoubleAdd.java | TestDoubleAdd.java

```
import junit.framework.TestCase;


public class TestDoubleAdd extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testDoublwAdd(){
        DoubleAdd number = new DoubleAdd();
        assertEquals(3.0, number.add(1, 2));
    }

    }
```

测试结果


Package Explorer
JUnit
Finished after 0.016 seconds
Runs: 1/1
Errors: 0
Failures: 1
TestDoubleAdd [Runner: JUnit 3]
testOver
Failure Trace
junit.framework.AssertionFailedError: expected:<3> but was:<3.0>
at junit.framework.Assert.fail(Assert.java:47)
at junit.framework.Assert.failNotEquals(Assert.java:280)
at junit.framework.Assert.assertEquals(Assert.java:64)
at junit.framework.Assert.assertEquals(Assert.java:71)
at TestDoubleAdd.testOver(TestDoubleAdd.java:14)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
at junit.framework.TestCase.runTest(TestCase.java:164)
at junit.framework.TestCase.runBare(TestCase.java:130)
at junit.framework.TestResult$1.protect(TestResult.java:106)
at junit.framework.TestResult.runProtected(TestResult.java:124)
at junit.framework.TestResult.run(TestResult.java:109)
at junit.framework.TestCase.run(TestCase.java:120)
at junit.framework.TestSuite.runTest(TestSuite.java:230)
at junit.framework.TestSuite.run(TestSuite.java:225)
at org.eclipse.jdt.internal.junit.runner.junit3.JUnit3TestReference.run(JUnit3TestReference.java:130)
at org.eclipse.jdt.internal.junit.runner.TestExecution.run(TestExecution.java:38)
at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:460)
at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.runTests(RemoteTestRunner.java:673)
at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run(RemoteTestRunner.java:386)
at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main(RemoteTestRunner.java:196)

小结

- **本章主要讲解了白盒测试的基本概念和技术，包括白盒测试的基本概念、分类、白盒测试中的边界值技术、语句覆盖测试、分支覆盖测试、条件覆盖测试、分支-条件覆盖测试、条件组合覆盖测试、路径覆盖测试…**
- **白盒测试允许观察“盒子”内部，不像黑盒测试那样把系统理解为一个“内部不可见的盒子”，不需要明白内部结构。**
  - 为了完整的测试一个软件，这两种测试都是不可或缺的。一个产品在其概念分析阶段直到最后交付给用户期间往往要经过各种静态的、动态的、白盒的和黑盒的测试。